Compilers

Dr. Sherin ElGokhy

Lecture#2

Lexical Analysis

Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexers
  - Regular expressions
  - Examples of regular expressions

The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

Lexical Analysis phase is the first phase of the long series implementation of compilers.

The Lexical Analyzer scans the input string to isolate its words and identify tokens.

Token = Lexeme = word (The basic building block in any language).

Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
    Z = 0;
else
    Z = 1;
```

- The analyzer will see this piece of code as just a string of characters with all white spaces symbols:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

Goal: Partition input string into substrings

- Where the substrings are tokens
- Token is a sequence of characters that have a collective meaning.

What's a Token?

- Lexical Analyzer does not just recognize the lexical units , it also classify these units according to their role
- A syntactic category
  - In English:

    noun, verb, adjective, …

  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …

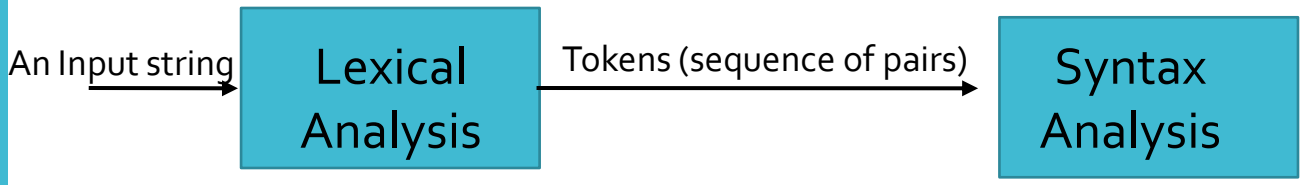
Tokens

- Tokens correspond to sets of strings.

- Identifier: *strings of letters or digits, starting with a letter*
- Integer: *a non-empty string of digits*
- Keyword: *"else" or "if" or "begin" or …*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

Lexical Analysis Goal

- Classify program substrings (tokens) according to role.
- Output of lexical analysis is a stream of tokens . . .
- . . . which is input to the parser



- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser

Example

- Recall

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- Useful tokens for this expression:

Integer, Keyword, Relation, Identifier, Whitespace, (,), =, ;

- ., (,), =, ; are tokens, not characters, here.
- Some single character token classes will be marked by themselves

Quiz

For the code fragment below,
choose the correct number of tokens in
each class that appear in the code fragment

```
x = 0;\n\twhile (x < 10) {\n\tx++;\n}
```

- W = 9; K = 1; I = 3; N = 2; O = 9
- W = 11; K = 4; I = 0; N = 2; O = 9
- W = 9; K = 4; I = 0; N = 3; O = 9
- W = 11; K = 1; I = 3; N = 3; O = 9

W: Whitespace
K: Keyword
I: Identifier
N: Number
O: Other Tokens:
{ } ( ) < ++ ; =

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token
- Recall:
  - Identifier: *strings of letters or digits, starting with a letter*
  - Integer: *a non-empty string of digits*
  - Keyword: *"else" or "if" or "begin" or …*
  - Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

Lexical Analyzer: Implementation

- An implementation must do two things:

1. Recognize substrings corresponding to tokens (lexemes)
2. Identify the token class of each lexeme.

Example

- Recall:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

Lexical Analyzer: Implementation

- The lexer usually discards "uninteresting" tokens that don't contribute to parsing.
- Examples: Whitespace, Comments

True Crimes of Lexical Analysis

- Is it as easy as it sounds?
- Not quite!
- Look at some history . . .

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- E.g., `VAR1` is the same as `VA  R1`
- A terrible design!

Example 1

- Consider
  - DO 5 I = 1,25
  - DO 5 I = 1.25
- An example of lexical analysis that requires **Lookahead.**
- One of the goals of lexical analysis is to minimize the amount of lookaheads or bound it to some constant to simplify the implementation of lexical analyzer.

Example 2

- Consider

```
if (i == j)
Z = 0;
else
Z = 1;
```

- Even this simple example has lookahead issues
  - i vs. if
  - = vs. ==
- Footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators

Lexical Analysis in FORTRAN (Cont.)

- Two important points:
  1. The goal is to partition the string. This is implemented by reading left-to-right , recognizing one token at a time
  2. "Lookahead" may be required to decide where one token ends and the next token begins

Lexical Analysis in PL/I

- PL/I keywords are not reserved: you can use a keyword both as a keyword and also as a variable name

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:

  DECLARE (ARG1,. . ., ARGN)

- Can't tell whether DECLARE is a keyword or an array reference until after the).
  - Requires lookahead!. Scan beyond this entire argument list to see what came next.

Lexical Analysis in C++

- Unfortunately, the problems continue today
- C++ template syntax:

```
Foo<Bar>
```

- C++ stream syntax:

```
cin >> var;
```

- But there is a conflict with nested templates:

```
Foo<Bar<Bazz>>
```

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token class of each lexeme
- Left-to-right scan => lookahead sometimes required

Next

- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is if two variables i and f?
    - Is == two equal signs = =?

Regular Languages

- There are several formalisms for specifying tokens
- We must say what set of strings belongs to a token class
  - Use regular Languages
- *Regular languages* are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Atomic Regular Expressions

- To define regular functions, we must define regular expressions.
- Single character

$$'c' = \{"c"\}$$

- Epsilon

$$\varepsilon = \{""\}$$

Compound Regular Expressions

- Union

$$A + B = \{ s \mid s \in A \text{ or } s \in B \}$$

- Concatenation

$$AB = \{ ab \mid a \in A \text{ and } b \in B \}$$

- Iteration

$$A^* = \bigcup_{i \geq 0} A^i \quad \text{where} \quad A^i = A \ldots i \text{ times} \ldots A$$

Regular Expressions

- **Def.** The *regular expressions R over* Σ are the smallest set of expressions including

$$\varepsilon$$

$$'c' \quad \text{where } c \in \Sigma$$

$$A + B \quad \text{where } A, B \text{ are rexp over } \Sigma$$

$$AB \quad " \quad " \quad "$$

$$A^* \quad \text{where } A \text{ is a rexp over } \Sigma$$

- Σ is the given alphabet (family of characters that form any regular expression).

Regular Expressions Quiz

Choose the regular languages that
are equivalent to the given regular language: $(0 + 1)^*1(0 + 1)^*$

$\Sigma = \{ 0, 1 \}$

- ☐ $(01 + 11)^*(0 + 1)^*$
- ☐ $(0 + 1)^*(10 + 11 + 1)(0 + 1)^*$
- ☐ $(1 + 0)^*1(1 + 0)^*$
- ☐ $(0+ 1)^*(0 + 1)(0 + 1)^*$

Review

- Regular expressions specify regular languages.
- Regular expressions have Five constructs
  - Two base cases
    - empty and 1-character strings
  - Three compound expressions
    - union, concatenation, iteration
- Regular expressions are simple, almost trivial
  - But they are useful!

- Reconsider informal token descriptions . . .

Languages

Def. Let Σ be a set of characters. A *language over* Σ is a set of strings of characters drawn from Σ

Examples of Languages

- Alphabet = English characters
- Language = English sentences

- Not every string of English characters is an English sentence

- Alphabet = ASCII
- Language = C programs

- Note: ASCII character set is different from English character set

Syntax vs. Semantics

- To be careful, we should distinguish syntax and semantics.

$$
\begin{aligned}
L(\varepsilon) &= \{""\} \\
L('c') &= \{"c"\} \\
L(A+B) &= L(A) \cup L(B) \\
L(AB) &= \{ab \mid a \in L(A) \text{ and } b \in L(B)\} \\
L(A^*) &= \bigcup_{i \geq 0} L(A^i)
\end{aligned}
$$

Syntax vs. Semantics

- Meaning function L maps syntax to semantics. A piece of syntax is converted to a set of strings.
- Meaning function L maps from expressions into sets.
- Why use a meaning function?
  - Makes clear what is syntax, what is semantics.
  - Allows us to consider notation as a separate issue
  - Because expressions and meanings are not 1-1
- Meaning is many to one: many different syntax pieces map to the same meaning.
  - Never one to many!

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want
- The standard notation for regular languages is *regular expressions.*

Example: Keyword

Keyword: "*else*" or "*if*" or "*begin*" or …

'else' + 'if' + 'begin' + . . .

Note: 'else' abbreviates 'e"l"s"e'

Example: Integers

Integer: *a non-empty string of digits*

digit = '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'

integer = digit digit*

Abbreviation: $A^+ = AA^*$

Example: Identifier

Identifier: *strings of letters or digits, starting with a letter.*

letter = 'A' + . . . + 'Z' + 'a' + . . . + 'z'

identifier = letter (letter + digit)*

Is (letter* + digit*) the same?

Example: Whitespace

Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

$$(' \ ' + '\n' + '\t')^{+}$$

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (650)-723-3232

$$
\begin{aligned}
\Sigma &= \text{digits} \cup \{\text{-},(,)\} \\
\text{exchange} &= \text{digit}^3 \\
\text{phone} &= \text{digit}^4 \\
\text{area} &= \text{digit}^3 \\
\text{phone\_number} &= \text{'('} \ \text{area} \ \text{')-'} \ \text{exchange} \ \text{'-'} \ \text{phone}
\end{aligned}
$$

Example: Email Addresses

- Consider *anyone@cs.stanford.edu*

$$\Sigma = \text{letters} \cup \{.,@\}$$

$$\text{name} = \text{letter}^{+}$$

$$\text{address} = \text{name '@' name '.' name '.' name}$$

Example: Unsigned Pascal Numbers

digit	=	'0' +'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'
digits	=	digit^+
opt_fraction	=	('.' digits) $+ \varepsilon$
opt_exponent	=	('E' ('+' + '-' + ε) digits) + ε
num	=	digits opt_fraction opt_exponent

Quiz

Choose the regular languages that are correct specifications of the English-language description given below:

Twelve-hour times of the form "04:13PM". Minutes should always be a two digit number, but hours may be a single digit.

- ☐ (0 + 1)?[0-9]:[0-5][0-9](AM + PM)
- ☐ ((0 + ε)[0-9] + 1[0-2]):[0-5][0-9](AM + PM)
- ☐ (0*[0-9] + 1[0-2]):[0-5][0-9](AM + PM)
- ☐ (0?[0-9] + 1(0 + 1 + 2):[0-5][0-9](A + P)M

Summary

- Regular expressions describe many useful languages
- Regular languages are a language specification
  - We still need an implementation

Thanks